XXXIII Неделя науки СПбГПУ. Материалы межвузовской научно-технической конференции. Ч.IX: С.71-72, 2005.

УДК 681.3

В.Р.Чурилов (асп., каф. информатики), А.Ю.Глебовский, к.ф.-м.н., доц.

ПОДСИСТЕМЫ ДЕКЛАРАТИВНОГО ВВОДА-ВЫВОДА В АППЛИКАТИВНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ.

Стандартные подсистемы ввода-вывода в современных языках программирования как аппликативных, так и императивных имеют низкоуровневый императивный интерфейс, что затрудняет процесс разработки несистемных приложений. Проиллюстрировать проблему можно с помощью фрагмента программы на функциональном языке Clean осуществляющего чтение bitmap-файла:

```
# (_, c1,file)  = sfreadc file
# (ok,c2,file) = sfreadc file      // прочитать два байта
  | c1<>'B' || c2<>'M'             // представляют ли они строку «BM»
      = abort ("File " +++ fileName +++ «not an BMP file»)
# (_,  fileSize, file)   = sfreadi file // прочитать длину
# (_,     _,       file)   = sfreadi file // пропустить 4 зарезервированных байта
# (_,     _,       file)   = sfreadi file // пропустить смещение
# (_,     _,       file)   = sfreadi file // пропустить длину заголовка
# (ok1, w,       file)   = sfreadi file // прочитать длину
# (ok2, h,       file)   = sfreadi file // прочитать высоту
```

Типичный набор действий в таком случае включает в себя цепочку стандартную цепочку операций (открытие файла, вызовов функций чтения данных с диска поэлементно, преобразование данных, присваивание преобразованных данных переменным программы, закрытие файла, обработка исключений и нестандартных ситуаций). Далее происходит непосредственное использование переменных, после чего результаты могут быть записаны обратно в файл, что подразумевает аналогичную цепочку действий. Такой подход к вводу-выводу в большинстве случаев является неоправданно низкоуровневым.

В данной работе предлагается использовать иной подход к вводу-выводу основанный не на низкоуровневых императивных операциях ввода-вывода, а на взаимном отображении переменных программы и байтового потока (дискового пространства либо данных передающихся по сети). Механизм отображения должен учитывать характер доступа к данным. Так если приложение потребует выборочный доступ к некоторой части данных, отображение должно осуществляться по необходимости (lazy mapping). В случае если приложению потребуется последовательный доступ к большей части данных, отображение должно осуществляться большими блоками оптимального размера для конкретной иерархии памяти (дисковый кэш – память – кэш первого и второго уровней). В API ввода-вывода некоторых операционных систем предусмотрена возможность специфицировать характер доступа к файловым данным, что существенно облегчает реализацию механизма отображения. Для работы с batmap-файлом можно определить следующее отображение:

```
mapping BMP  {
        2 X byte [r]: (char[2])            identifier
        4 * byte  [r]: (LE int)            size
(10)    4 X (4 * byte) [r]: (LE int)       data_offset, header_size, width, height
        2 X (2 * byte) [r]: (LE short)  n_planes, Bits_per_pixel
```

```
        6 X (4 * byte) [r]: (LE int)      compression, data_size, h_resolution,
                                          v_resolution, n_colors, important_colors
        /-> exception "Invalid header in $filename file"
        n_colors X (4 X (byte)) [r]: (2|1| byte[n_colors][4]) palette
        /-> exception "Invalid palette in $filename file"
(data_offset)
        data_size * byte [rw]: (2|1| byte [height][width]) bitmap <Sequential ReadWhole>
        /-> exception "Unexpected end of file $filename while reading its contents"
}
```

В основном разделе файла отображения каждое выражение (например «(10) 4 X (4 * byte) [r]: (LE Int) data_offset, header_size, width, height») определяет, как отображать набор байтов в набор программных переменных и состоит из ниже перечисленных компонентов.

1. Смещение относительно начала байтового потока (файла) – «(10)». В случае если смещение не указывается, оно подразумевается порядковой позицией выражения.
2. Количество однотипных блоков данных следующих подряд друг за другом – «4 X»
3. Количество байтов, занимаемое блоком данных в байтовом потоке – «(4 * bytes)»
4. Тип данных определяет тип данных языка программирования, в который осуществлять преобразование набора байтов – «(LE int)». При этом указывается порядок байтов (LE – Little Endian, BE – Big Endian).
5. Спецификатор доступа – Доступ на чтение, запись или чтение/запись – «[r]:».
6. Имена переменных, каждая из которых будет отображаться в соответствующий блок данных (набор байтов) – «data_offset, header_size, width, height».
7. В случае массивов данных отображение может задавать преобразование потока байтов в массив заданной размерности. Здесь же указывается порядок следования элементов массива в байтовом потоке (так для двумерного массива возможно хранение по строкам – «2|1|» или по столбцам – «1|2|»).
8. Может определяться характер доступа к данным – «<Sequential ReadWhole>».

Преимущества предлагаемого подхода перечислены ниже.

1. Разделение операций ввода-вывода и логики приложения.
2. Меньшее количество кода в сравнении с традиционным подходом. Отображение – декларативная и более высокоуровневая конструкция, позволяющая, тем не менее, оперировать достаточно универсальной конструкцией – потоком байтов.
3. Оптимизация «под ключ». Каждое приложение может указать какой характер доступа к данным (data access pattern) оно использует – например последовательный, случайный, выборочный.

Для реализации предлагаемой схемы ввода-вывода можно создать специальный тип файла, который будет содержать спецификацию отражения. Для функциональных ЯП это требует поддержания прозрачности ссылок (referential transparency). В императивных ООЯП таких как Java это может быть реализовано генерацией подкласса в котором каждый доступ к переменной класса-предка созданного разработчиком будет перехватываться подклассом, в котором и будет реализован механизм отображения.